1983

# A Critical Analysis of the Design Features Supported by the Systems with ADA

Sunil Agarwal
*University of Rhode Island*

# A CRITICAL ANALYSIS OF THE DESIGN FEATURES SUPPORTED BY THE SYSTEMS WITH ADA

BY

SUNIL AGARWAL

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

1983

MASTER OF SCIENCE THESIS

OF

SUNIL AGARWAL

Approved:

Thesis Committee

Major Professor _____

_____ 8/11/83

_____

_____

Dean of the Graduate School

UNIVERSITY OF RHODE ISLAND

1983

## ABSTRACT

Certain simplifications in the programming language Ada are suggested. The simplifications include the removal of derived types and minimization of intertask communication in multicomputer like environments to simplify the language design. Suggestions are also made to improve the expressiveness and capabilities of Ada by introducing subprogram types, association of implicit queues with entry points of a task, redefinition of the assignment operator and a mechanism for descheduling tasks. These could lead to improve run-time efficiency, exception handling mechanism an to achieve simplicity in the language design.

## ACKNOWLEDGEMENT

I take this opportunity to thank my major professor Dr. Jin W. Soh for providing valuable suggestions and encouragement throughout the development of this thesis. I would also like to thank Dr. Lamagna and Dr. Rajan for helping me with those "out of market" books on Ada and other references. A special thanks goes to Mr. Mark Gerhardt of Raytheon Corporation a SUBSIG division of portsmouth, RI for sharing his valuable time with me to point out some problems with Ada. Last, but not least, I thank my thesis committee for their suggestions and interest in the present work.

Table of contents

# INTRODUCTION

Recent advances in electronics have dramatically shifted the overall cost structure of computer systems from the cost of hardware to the cost of software development. Part of the reason is that programming is a labor intensive activity with complexities far beyond the capabilities of a single programmer. Thus it is not unreasonable to expect programs, with a typical life span of 10 years, being written by a team of people with several man years of involvement. This calls for breaking of a complex problem into independent modules, communication between programmers to know about the assertions each one makes about his module, and finally putting the modules together to obtain a unified 'working' program. Changes in a program, also called maintenance, are inevitable over the span of its life. A study [6] has shown that such maintenance costs exceed the development costs of the original program. The main reasons which contribute to high maintenance costs are as follows:

(1) A turnover in the original staff. Thus a cost is associated with training a new programmer.

(2) The structure of the final program is of little help in understanding how the original problem was decomposed and programmed. This adversely affects the precise understanding of the program. Though informal documentation, through comments, can be provided, it leaves enough room for ambiguities and

misinterpretation.

(3) Changes during the maintenance phase are liable to cause impairment of the original structure of the programs thereby making future changes increasingly difficult to introduce.

E J Dijkstra wrote about this software crisis [5]

" As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now that we have gigantic computers, programming has become an equally gigantic problem. In this sense the electronic industry has not solved a single problem, it has only created them, it has created the problem of using its products ".

## 1.1 DESIRED FEATURES IN PROGRAMMING LANGUAGES

The present day software technology emphasizes the need of programming languages which

- Facilitate easy writing and reading of a program.
- Retain the decompositional structure of the original problem into the final program.
- Provide facilities for developing modules of a complex problem independently.
- Facilitate communication among members of a team working on some global project by providing ways to specify an interface of a module to its external environment without being concerned with its implementation details.

- Provide ways to specify concurrent execution to exploit the availability of cheaper microprocessors and supporting hardware components.
- Provide facilities to specify low level details in a given application to generate efficient code.

Motivated by these requirements, the US Department of Defense launched a project to design a language and its support environment to incorporate these to simplify software development. The present thesis is based on the outcome of this massive effort, the language Ada.

## 1.2 MOTIVATION BEHIND THE PRESENT STUDY

There always seems to be a resistance in the software community to accepting a new language. No matter how reasonable this resistance may sound, due to the growing technology and the simultaneous growth of the application domain, earlier languages like FORTRAN, PASCAL, PL/I can not be used to meet the new challenge in the right perspective. These languages have been modified over the years, eg. FORTRAN77, Concurrent PASCAL to keep up with the application needs but without much impact on programming style. These languages, as pointed out earlier also, have several drawbacks, which prevent them having any future in the development of complex software projects. The language Ada, in the beginning, was designed keeping the requirements of

embedded systems [6] in mind but it is powerful and general enough to replace basically all existing higher level languages. To drive the idea home, let us compare the language, PASCAL, ( pointing out PASCAL's drawbacks through an example) with the language Ada and see how these problems are resolved in Ada.

1.2.1 sample problem

A database containing information about employees, including their names, divisions, telephone numbers, and salaries is assumed to be available. The program must produce a data structure containing a sorted list of the employees in a selected division and their telephone numbers.

A typical PASCAL solution may look as follows:

```
C Defining all the useful types
C for the program
type     STRING = packed array [1..24] of CHAR;
C type STRING is for the name.
C Name can be 24 characters long.
         SHORTSTRING = packed array [1..8] of CHAR;
C type SHORTSTRING is for the name of the division.
C Division name can be eight characters long.
C Information about each employee is available in
C the records of type EMPREC.
         EMPREC = record
           NAME: STRING;
```

```
        PHONE: INTEGER;
        SALARY : REAL;
        DIVISION: SHORTSTRING;
      end;
C Result is returned in the variable
C of type PHONEREC.
      PHONEREC = record
        NAME: STRING;
        PHONE: INTEGER;
      end;
C Declaring program variables.
var   STAFF: array [1..100] of EMPREC;
      PHONE: array [1..100] of PHONEREC;
      STAFFSIZE, DIVSIZE, I, J: INTEGER;
      Q: PHONEREC;
begin
C Size of the division of interest is
C initialized to zero.
  DIVSIZE := 0;
C In this loop the names and phone numbers
C of all the employees belonging to the division
C denoted by WHICHDIV are collected in array PHONE.
  for I:=1 to STAFFSIZE do
    if STAFF[I].DIVISION = WHICHDIV then
C WHICHDIV corresponds to the division of interest.
      begin
        DIVSIZE := DIVSIZE + 1;
```

```
        PHONES[DIVSIZE].NAME := STAFF[I].NAME;
        PHONES[DIVSIZE].PHONE := STAFF[I].PHONE;
      end;
C within these loops the array PHONE is sorted
C with respect to employees names.
  for I:=1 to DIVSIZE do
    for J:=I+1 to DIVSIZE do
      if PHONES[I].NAME ≥ PHONES[J].NAME then
      begin
        Q := PHONES[I];
        PHONES[I] := PHONES[J];
        PHONES[J] := Q;
      end;
end;
```

The problems with PASCAL (ignoring the aspects of concurrent programming and low level manipulations) are

(a) It provides limited support for large programs and is lacking in separate compilation facilities and block structure other than nested procedures.

(b) There is no support for the encapsulation of related definitions in such a way that they can be isolated from the remainder of the program.

(c) It does not allow programmer defined types to accept parameters. If it did then we could have done with only a single definition for both types STRING and SHORTSTRING by passing it a length parameter in the

above program.

(d) It does not provide any information hiding. For example, there is no way to make the name of an employee known without making his salary (a sensitive datum) also known.

The above example can be programmed in a much better way using Ada as follows:

```
package EMPLOYEE is
type PRIVSTUFF is limited private;
type EMPREC is
   record
     NAME: STRING(1..24);
     PHONE: INTEGER;
     PRIVPART: PRIVSTUFF;
   end record;
procedure SETSALARY (WHO: in out EMPREC; SAL:FLOAT);
function GETSALARY (WHO:EMPREC) return FLOAT;
procedure SETDIV (WHO: in out EMPREC; DIV: STRING(1..8));
function GETDIV (WHO: EMPREC) return STRING(1..8);
private
   type PRIVSTUFF is
     record
       SALARY: FLOAT;
       DIVISION: STRING(1..8);
     end record;
end EMPLOYEE;
```

Note that in this package we have been able to encapsulate

the definition of type EMPREC and its associated operations within the package EMPLOYEE. Sensitive data like salary and division are declared to be private. Thus they can only be accessed through subprograms, which may check the user-id before allowing the access, the access being defined in the visible part of the package. The package body may be defined and compiled later but prior to execution. Since the type STRING is parameterized, we do not have to define two separate types for NAME and DIV. The main program is almost identical to the PASCAL program, except for some minor syntax differences, hence not discussed. In brief, the language Ada not only supports the capabilities of existing languages, but facilities for separate compilation, top-down and bottom-up program development, data protection, real time programming among others. It is predicted that Ada will dominate the programming activities in late 80's and 90's. It appears therefore worthwhile and significant to study if Ada indeed does what it promises in application programming.

## 1.3 OBJECTIVE OF THE STUDY

The design of systems utilizing the complete set of capabilities is quite complex. To date there has not been a single such system even though the syntax of the language were first published in 1980. A major reason for this delay appears to be that the designers of the

language failed to understand the depth of complexities in its implementation. In this thesis some of the problems of Ada are identified and some simplifications are suggested. same level of functionality and expressiveness (Chapter Two).

An attempt is also made to improve some aspects of Ada by introducing new concepts. These concepts add to the expressive power, run-time efficiency and to the overall capabilities of the language.

# CHAPTER TWO

## SUGGESTIONS TO SIMPLIFY ADA

The meaning of 'simplicity' when applied to programming languages can be quite deceptive. In the hierarchy of programming languages, assembly language is conceptually very simple. But writing an average size program in assembly language can be quite cumbersome. What assembly language lacks is expressiveness. A language is expressive if it has a built-in abstract mechanism to simplify the programming of a problem at hand e.g. a variable name is an abstraction of a memory location. It can be made increasingly expressive by adding special-purpose abstraction. Therefore a language can be very simple without being expressive. On the other hand, it can be made very expressive but not so simple to implement.

The aim of this chapter is to identify and remove those abstractions in Ada whose functionality can be realized using other built-in abstractions in the language. This will simplify the language design without any significant loss in its expressiveness and portability. The discussion is limited to two aspects of Ada. Namely,

(1) Derived Types,

(2) Concurrency on Multicomputers.

### 2.1 DERIVED TYPES

A type in Ada can be derived from a parent type such that the derived type inherits some of the operations defined for the parent type including all basic and predefined operators and the domain of the derived type is same as its parent type. A compiler sees them as two distinct types, interconvertible by explicit type conversions [19]. Here we shall attempt to show that this facility is superfluous and the same functionality can be achieved using other features available in the language thus reducing unneccessary complications in the language design. The following is a list describing the complications introduced in the language design by the derived type definition

(a) Explicit conversion operators have to be made available by the compiler.

(b) Each basic and predefined operator of the parent type have to be redefined for the derived type.

(c) Certain subprograms that define the operations of the parent type are to be redefined implicitly for the derived type in a transitive manner.

(d) Overloaded literals, aggregates, entry, slice, attributes, etc. have to be resolved for the derived and parent types.

Implicit definitions slow down the compilation process while making the design of compiler more difficult. The Ada language design team has listed four motivations [13] for the derived type facility as follows:

(1) Achieving portability of programs using numeric types.

(2) Poor man's strong typing

(3) Change of representation: if the derived type T is explicitly given a representation specification (indicating offsets and widths of record fields and other low level details), then conversion between T and its parent type also effects a change of representation.

(4) Enable the construction of private types.

The following discussion, based on a paper by Hilfinger [13], will illustrate how to achieve the same level of capabilities without resorting to derived types.

### 2.1.1 PORTABILITY OF NUMERIC TYPES

The type

```
type INT is L..U;
```

is elaborated using a derived type definition as follows:

```
type INTEGER-TYPE is new PREDEFINED-INTEGER-TYPE;
subtype T is INTEGER-TYPE range <>
INTEGER-TYPE(L)..INTEGER-TYPE(U);
```

where INTEGER-TYPE is an anonymous type and PREDEFINED-INTEGER-TYPE is implicitly selected by the compiler so as to contain the values L and U (inclusive) thus providing complete portability [19]. A similar elaboration definition holds for the real type.

To achieve the same effect, without derived types, we can define

subtype INT is INTEGER-TYPE range L..U;

where the programmer has to select INTEGER-TYPE depending upon the implementation. The point to note here is that only a subset of these subtype definitions need be changed when moving to a new target machine because most of the defined INTEGER-TYPE will satisfy the range constraint. The compiler will check this compatibility in relation to the target machine. Additional advantages of this approach are

(1) Subtype definitions do not introduce a new type; thus the compiler does not have to manage too many types.

(2) The management of mathematical packages is simplified. For example, suppose we want to use two independently compiled generic mathematical packages in a program. Assume that these packages are made generic in type REAL. This will cause a communication problem when the results are to be exchanged between these two packages since the type in each package is distinct. The programmer has to perform explicit type conversions to exchange the results. A better solution is to define a mathematical package as follows:

```
package MATH-FUNCTIONS is
function SIN(X:FLOAT) return FLOAT;
function SIN(X:LONG-FLOAT) return LONG-FLOAT;
end MATH-FUNCTIONS;
```

In the above package, function SIN is provided for

all possible real types supported by the machine. The communication problem goes away because now mathematical packages need not be made generic in type REAL.

### 2.1.2 POOR MAN'S STRONG TYPING

The Ada compiler prohibits the mixing of mathematically identical but unrelated data types. This can prove helpful in eliminating unintended programming errors. For example,

```
type DOLLARS is new INTEGER;
type POUNDS is new INTEGER;
```

The compiler will prohibit mixing of POUNDS with DOLLARS. This strong type checking can't be provided without derived types. On the issue of desirability of such strong type checking in Ada, Hilfinger points out [13]

" This type checking was a side effect of the attempt to achieve the portability of programs rather than to a perceived need to provide a way of differentiating arithmetically identical numeric types".

Thus it is hardly justified to introduce derived type for this sort of checking.

### 2.1.3 CHANGE OF REPRESENTATION

Ada provides a facility for specifying the hardware implementation of a type through representation clauses.

At most one representation clause is allowed for a given type. To illustrate how to achieve the change of representation, consider the following example

```
type A is
  record
    AGE:INTEGER;
    WEIGHT:INTEGER;
  end;
```

Now using derived type definition we define another type B

```
  B is new A;
```

and impose the following restriction on the representation of B

```
for B use
  record
    AGE at 0*BYTE range 0..3;
    WEIGHT at 0*BYTE range 4..7;
  end;
```

Now the declaration

```
X:A; Y:B;
```

defines two record variables with two different representations. A change of representation can be obtained by a simple assignment statement as follows:

```
    Y := B(X);
```

Note that type-mark B is required to effectuate the change of representation. To achieve the same effect without derived types, let us define the type B as

```
type B is
```

```
record
   AGE:INTEGER;
   WEIGHT:INTEGER;
end;
```

and its representation clause as defined before. The conversion from one representation to another can be achieved by a user defined procedure CONVERT as indicated below

```
procedure CONVERT (ORIGINAL-REP:A; DESIRED-REP: out
B);
   begin
   DESIRED-REP.AGE := ORIGNAL-REP.AGE;
   DESIRED-REP.WEIGHT := ORIGINAL-REP.WEIGHT;
   end CONVERT;
```

Thus all we need to effectuate a change in representation is to call the procedure CONVERT as follows

```
CONVERT(X,Y);
```

2.1.4 PRIVATE TYPES

Private types is a way to allow limited access to a variable of this type thus protecting it from undesirable/unintended operations.

Consider the following package specifications:

```
package A is
type INT is private;
function "+" (U,V: INT) return INT;
private
```

```
type INT is new INTEGER;
end A;

package body A is
function "+"( U,V:INT) return INT is
  begin
    return INT'(INTEGER(U) + INTEGER(V));
  end;
end A;
```

The derived type enables a programmer to define a type INTEGER which is distinct from the predefined type INTEGER syntactically. The same functionality can be achieved without derived types by elaborating the private type definition as

```
private
type INT is
  record
    X: INTEGER;
  end;
```

and defining the package body as

```
package body A is
function "+" (U,V:INT) return INT is
  begin
    return INT'( U.X + V.X);
  end;
end A;
```

Here the record type elaboration of the private type behaves exactly as the previously defined private type as far as the user of this package is concerned.

This demonstrates very convincingly that the introduction of derived types is an unnecessary complication of the language as the desired functionality can be achieved using existing facilities in the language.

## 2.2 CONCURRENCY ON MULTICOMPUTERS

The Ada reference manual states that " tasks can be implemented on multicomputers, multiprocessors or with interleaved execution on a single processor." Let us look at the implementation aspects of tasks on multicomputers, i.e. a computer architecture consisting of several different computers without shared memory. The communication between these computers is done by sending messages to each other. Two tasks being executed on different computers in Ada may

(a) not communicate at all with each other;

(b) communicate through global variables only; and,

(c) communicate through entry calls and global variables.

Implementation of case (a) is very straight forward but the latter two cases will require much additional communication overhead. Consider two tasks A and B being executed by different computers, and operating on the same global variable G. The variable G may reside in the memory of computer A or B or possibly in another computer,

say C. In the worst case assume that it resides in the memory of computer C. This implies that every access to G from task A or task B will require communication with computer C. Such communication protocols will have to be provided at compilation time thus making the language design more complex. Similarly when global variables are access variables, then again extra communication overhead will be required. Note in this case the pointer values will have to indicate which computer's memory they are pointing to. Thus the pointer management will become more complex and its size will possibly have to be increased to identify the computer whose memory is being pointed to. These inter-computer communications besides adding to the complexity of the design, affect the run-time efficiency adversely.

The Ada manual [19] specifies the following for the access of global variables by two or more tasks.

" if two tasks read or update a shared variable then neither of them may assume anything about the order in which other performs its operations, except at the points where they synchronize. Tasks are synchronized at the start and at the end of their rendezvous and at the start and end of their activations".

The above specification can be used to draw the following two conclusions

(i) If, between two synchronization points of a task, a task reads a shared variable whose type is of the

scalar or access type, then the variable is not updated by any other task at any time between these two points.

(ii) If, between two synchronization points of a task, a task updates a shared variable whose type is scalar or access, then the variable is neither read nor updated by any other task between these two points.

These conclusions allow an implementation to keep a local copy of the shared variable in a register or memory of computer. This minimizes the intercomputer communication as it is required only at the start and at the end of the synchronization points. The problem of minimizing the communication overhead for those tasks that only communicate through global variables still remains. Note that an indivisible write or read operation on a shared variable can be provided by a pragma as follows:

pragma SHARED (simple-variable-name);

CAR Hoare [11] avoided this problem in his paper " Communicating sequential processes" by restricting the inter-task communication by means of input/output only that is through entry statements only.

There seems to be no nontrivial solution to take care of this problem unless some very efficient inter-computer communication protocol is available. As Gehani [8] points out, the implementation of Ada on multicomputers can be simplified by prohibiting task communication via global or access variables or by ensuring that tasks that share data

by means of global or access variables will reside on the same computer.

In conclusion, the removal of derived types and the imposing of the above restriction in implementing tasks on multicomputers potentially offer a substantial simplification in the language design. A little loss of portability and generality of Ada seems to be a small price to pay for the simplicity and efficiency it provides to the implementer.

# CHAPTER THREE

# IMPROVING ABSTRACT MECHANISMS IN ADA

Ada provides a rich set of abstractions to enhance its capabilities for a variety of applications such as systems programming, parallel programming, programming of embedded systems with real-time constraints and so on. There are situations when these abstractions are too general or too restrictive to satisfy the requirements of the problem. In this chapter we will concentrate on the following four situations to demonstrate the limitations in Ada.

(1) Message passing between tasks

(2) Redefinition of primitive operators

(3) Flexibility in scheduling discipline

(4) Exception Handling

An attempt will be made to eliminate these limitations by suggesting some improvements in the abstraction mechanism. As we shall see later, it will be achieved without making the design of the language any more complex.

## 3.1 MESSAGE PASSING BETWEEN TASKS

The design of systems with Ada is intended tc serve as a programming standard for embedded computer applications (i.e. command, control, communications, avionics, shipboard applications, etc.). As a consequence of its projected applications, the language contains facilities for parallel and real-time programming in multiprocessor

and multicomputer environments. Multiprocessor applications tend to have much more stringent requirements for run-time efficiency than do most applications developed for uniprocessor environment. Multiprocessor based systems have significant advantages over coventional uniprocessor environments in three distinct areas [26]

(1) Multiprocessor systems are capable of increased, effective throughput because they allow independent tasks within the application to operate in parallel.

(2) Multiprocessor systems can be designed to include software reliability structures that exploit the redundancy in hardware to dynamically alter the system configuration in response to hardware failure. Recovery from failure is a prime consideration in applications where human lives are involved such as monitoring nuclear plants or controlling war-weapons.

(3) Multiprocessor systems can be expanded gracefully as the requirements of the application domain change.

Among these, run-time efficiency seems to have favored the design of multiprocessor systems strongly. For this reason, the parallel control features provided by an implementation language intended for the use with multiprocessors must be designed to allow highly efficient

interprocess communication and control. In this section, we will analyse the suitability of Ada for multiprocessor applications in relation to the above requirement. Let us consider a typical application of a unidirectional communication between PRODUCER and CONSUMER tasks through a BUFFER task. Task BUFFER can be specified as follows

```
task BUFFER is
entry READ (M: out MESSAGE);
entry WRITE (M: MESSAGE);
end BUFFER;

task body BUFFER is
N: constant INTEGER :=10 1;
type INDEX is INTEGER range 1..N;
Q: array (INDEX) of MESSAGES;
FIRST,SECOND: INDEX :=1;
                loop
                  select
                   when LAST mod N + 1/= FIRST
                    accept WRITE (M: in MESSAGE) do
                    Q(LAST mod. N + 1) := M;
                    end WRITE;
                    LAST := LAST mod N + 1;
                  or
                   when FIRST /= LAST
                    accept READ (M: out MESSAGE);
```

```
                    M := Q(FIRST mod N + 1);
                    end READ;
                    FIRST := FIRST mod N + 1;
                end select;
            end loop;
    end BUFFER;
```

Now assume that the PRODUCER task executes the entry call

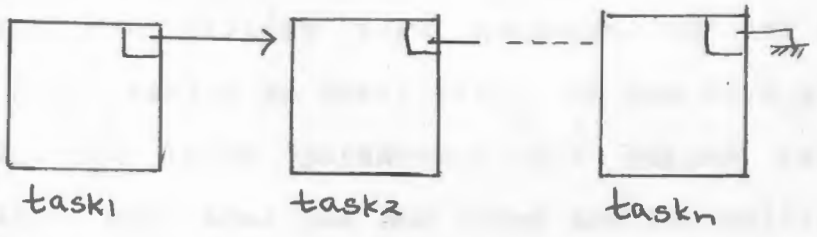
```
BUFFER.WRITE (SOME-MESSAGE);
```

to initiate the transfer. According to the semantics of the entry call, the PRODUCER task is now blocked until the BUFFER task is scheduled and completes the rendezvous. During this time the PRODUCER task must wait to be rescheduled when the BUFFER task completes the rendezvous. Thus, before a PRODUCER is allowed to proceed, two scheduling operations must occur. Similarly two more scheduling operations are required to read a message. This implies that a total of four scheduling operations are needed to transmit a single message. During this time PRODUCER or CONSUMER tasks are waiting without doing any useful work.

Since each scheduler interaction may involve a complete context swap, this implementation of message passing would be prohibitively expensive for many applications. To give an idea about the severity of the problem, if scheduling interactions are required to ensure the mutual exclusion,

the path through the critical region would require typically about 200 instructions there-by reducing the overall efficiency by an order of magnitude. If the critical region is sufficiently small, say 10 instructions, then the price to pay for ensuring mutual exclusion is intolerably high. Further damage to the efficiency of a real time program can result if this program requires frequent message passing between tasks, which is generally the case.

In an attempt to solve the above problem, let us look at how the the tasks are queued at an entry (say E) in a given task. Queues of tasks can be very easily implemented by reserving a queue pointer cell in the activation record of each task as shown in the following figure.




When a task calls entry E, it will be queued as shown, or when the called task executes the accept statement for entry E, then the first task (in this case, Task1) will be taken out from the queue. So basically this queue can be accessed by more than one task at a given time. The underlying system for Ada must ensure the indivisible

access to this queue by each task. Jim Welsh [26] suggested to extend the abstract mechanism in Ada to provide a data queue associated with an entry point. For the sake of simplicity assume the storage for the data queue is statically determined at compile time. As the mechanism to provide indivisible access to queues of tasks already exists, it is very easy to extend this to the data queue without adding to the complexity of language design. The modified BUFFER task can be defined as follows:

```
task BUFFER is
entry WRITE ((100)) (M: in MESSAGE);
entry READ ((100)) (M: out MESSAGE);
end BUFFER;
```

Without introducing any changes in the body of the task BUFFER. The parameter ((100)) specifies the data queue size. Now for the case of an entry which has only 'in' parameters, the calling task performs one of the two actions when making an entry call. If the data queue is not full, the input parameters are copied into the preallocated data area for the queue and the calling task is allowed to proceed even if BUFFER task is unable to complete the rendezvous. If the data queue is full then the calling task (PRODUCER) is queued at the appropriate entry (WRITE) as in the conventional approach. The BUFFER task, upon reaching an accept statement (WRITE in this case) performs operations in the following sequence:

(1) If the associated data queue is empty then do nothing; otherwise, copy the first data from the associated data queue into the local data area 'Q' associated with the task BUFFER.

(2) If, as a part of the same operation, the data queue was previously full, then pick the first task, if any, waiting at the entry and copy the associated message in the data queue. At this point the calling task is free to proceed.

A similar mechanism can be used to handle the case of entries which operate in the opposite direction and have only an 'out' parameter. Note that this implementation is possible because the PRODUCER and CONSUMER tasks have either all 'in' parameters or all 'out' parameters. This scheme will add to the run-time effiency by reducing the number of scheduler interactions involved. One interesting obeservation about the approach is that the data queue implementation can be interpreted as a pragma, which a compiler is free to ignore.

## 3.2 REDEFINITION OF PRIMITIVE OPERATORS

One of the main criticisms of Ada arises from the fact that it does not allow the redefinition of primitive operators, most notably the assignment operator. There are cases when redefinition of assignment and equality operators is desired. We will see that these

redefinitions add to the simplicity of design and expressiveness of Ada without affecting its run-time efficiency or the complexity of the compiler [13]. To illustrate the point, let us consider operations on varying length string variables. A type variable-string can be defined as follows:

```
subtype INDEX is INTEGER range 0..INTEGER'LAST;
type VSTRING(MAXLEN:INDEX) is
record
POS:INDEX :=0;
VALUE: STRING(1..MAXLEN);
end;
```

Following are the desirable operations on the variables of type VSTRING (PL/I provides these)

(a) V1 := V2 --copy a string variable

(b) V1 := "INITIALIZE TO STRING"

(c) V1 := V1 & "APEND";

According to the definition of assignment operator in current Ada [19], statement (a) will be legal if and only if V1 and V2 have the same maximum length. What is required is the desirable condition for the legality of the operation to be

length(V2) ≤ length(V1)

Statement (b) is not legal at all since the items to the left and right of the assignment operator are of two different types.

Statement (c) will be legal only if the variable returned

after concatenation satisfies the criterion mentioned in (a).

The desired effect can be achieved by defining a procedure such as

```
procedure SET(OBJECT: in out VSTRING; VALUE:STRING);
--------- body of the procedure
end SET;
```

which converts a string literal to a variable of type VSTRING.

But consider the following case:

```
procedure RANDOM(Z:VSTRING) is
---------- body of the procedure
end RANDOM;
```

As statements of type (b) are illegal, a call to procedure RANDOM

```
RANDOM("SOME STRING");
```

will be illegal because it involves an assignment of a string literal to a variable of type VSTRING. The way to get around this problem is to call procedure SET to convert the literal string to the variable of type VSTRING and then to call procedure RANDOM. Clearly it is an unnecessary complication in calling the procedure.

Things can be simplified if redefinition of the the assignment operator is allowed. A natural consequence of this will be that the equality operator "=" will also become an ordinary relational operator. The following is

a list of major changes, suggested by Hilfinger [13], to be done to introduce ":=" operator in the language.

(1) Remove the limited keyword and instead make all private types "limited" in the sense that they do not automatically export a ":=" or "=" operator.

(2) Add ":=" as a definable binary operator and make the operator "=" an ordinary relational operator in all respects.

(3) Define all operators ":=", "=", and "/=" implicitly on all scalar, access and composite types as provided in current Ada. Obviously assignment on composite types will be defined component-wise.

(4) Define a invocation of a ":=" operator to be type valid if two actuals have the same types as their corresponding formals. An invocation of any other subprogram is type valid if each actual parameter has the same type as the corresponding formal or is assignment compatible with the formal.

(5) Unconstrained formal parameters inherit any discriminant constraints as in current Ada except in the case where an unconstrained 'in' parameter does not have the same type as its corresponding actual. (We have seen this in

relation to procedure RANDOM). In this case, the discriminants of the formal are set by the assignment operator.

The following package can be defined, using := operator definitions, as a library unit to implement type VSTRING

```
package STRING is
type VSTRING (MAXLEN:INDEX) is private;
-- the domain of VSTRING(N) is all
-- strings with length ≤ N
procedure ":="(LHS: out VSTRING; S:STRING);
procedure ":="(X: out VSTRING; Y:VSTRING);
--copies Y into X iff the string in Y
--is in the domain of X
function "="(X,Y:VSTRING) return BOOLEAN;
function " & " (X,Y:VSTRING) return VSTRING;
private
type VSTRING(MAXLEN:INDEX) is
record
        POS:INDEX :=0;
        VALUE:STRING(1..MAXLEN);
end record;
end STRING;
```

Now the statements (a), (b), and (c) are legal if the string in RHS is in the domain of the string in LHS. And also the procedure RANDOM can be activated without any complications.

The redefinition of ":=" operator can also improve the expressiveness when dealing with literals. Suppose we wish to represent a set of integers. In current Ada one can write

```
subtype INT is INTEGER range 1..20;
type SET-OF-INT is array (INT) of BOOLEAN;
```

Now a set of integers (1,7,10,13) can be denoted as

```
SET-OF-INT'(1/7/10/13 => TRUE, others => FALSE);
```

A better notation would be

```
type SET-LITERAL is array (INTEGER range <> ) of INT;
procedure ":="(LHS: out SET-OF-INT; RHS:SET-LITERAL) is
begin
  LHS := SET-OF-INT'( others => FALSE);
  for I in RHS'range loop
  LHS(I) := TRUE;
  end loop;
end.
```

This enables us to represent set (1,7,10,13) as

```
   X:SET-OF-INT := (1,7,10,13);
```

This representation is quite expressive and conceptually simple. A redefinition of the assignment operator poses some validation problems. To illustrate the point, consider the following case:

```
   procedure ":=" (LHS: out INTEGER; RHS:INTEGER) is
   begin
  STANDARD."=" (LHS,0);
```

end;

Now this will always assign a integer value '0' to LHS thus the normal meaning of assignment operator has been breached. This will certainly make the programs more difficult to understand. The arguement for allowing this type of problem is that the definition of ":=" is a subprogram like any other and thus should be tested for the legality of the operation.

### 3.3 FLEXIBILITY IN SCHEDULING DISCIPLINE

One area in which Ada has been criticized widely is that it does not provide adequate control over scheduling policy [26]. To get the flavor of the problem, suppose Ada is chosen as the implementation language for the design and development of a time-sharing system for a multiprocessor architecture. Individual user processes can be represented as independent tasks in the time-sharing structure. In a typical time-sharing system, each user process is allowed execute for a limited time slot(quantum) in one stretch. If this slot is exceeded, the process is forcibly descheduled and other waiting processes are allowed to execute. The performance of a time-sharing system is quite sensitive to the size and dynamic behaviour of this quantum limit and it is important to be able to adjust this to conform to the loading demands.

In Ada, there is no apparent way to specify a run-time limit for a task nor is it possible for one task to control the scheduling or descheduling of the other tasks. According to the present semantics in Ada, two indirect approaches are possible for scheduling the tasks.

(1) Design a scheduler which operates cooperatively in the sense that task themselves participate in scheduling decisions. In this case, each task would be required to check periodically its accumulated run-time and dismiss itself by executing a delay statement.

(2) Provide interrupt entries in each task. The scheduler will generate a software interrupt for the associated entry of the task to be deallocated. This task can deschedule itself by executing a delay statement within its corresponding entry.

The first approach can be rejected outright because it requires the compiler to perform complex path analysis and to assemble the code to poll the scheduler at frequent intervals.

The second alternative may seem acceptable but it is an implementation dependent feature. Thus portability is sacrificed and moreover the readability of the program is

made difficult.

Jim Welsh [26] proposed solving this potential problem by introducing a language primitive

deschedule T;

The effect of this statement will be to deschedule the task pointed to by the task object T, thus allowing other waiting tasks to execute. A second alternative, according to the author, can be to provide a provisicn in Ada to alter the priority of the tasks dynamically. In current Ada the priority of a task is fixed at the compile time. As the run-time system schedules the tasks with higher priority first, the desired effect can be achieved.

## 3.4 EXCEPTION HANDLING

Traditionally exceptions are the error conditions that arise during program execution when certain operations are invoked. Bringing an exception condition to an invoker's attention is called raising an exception. The invoker responds to this exceptional condition through a corresponding handler. A few examples of exceptions are overflow exception, underflow exception, and end of file exception. The trend in programming languages these days is to be able to raise exceptions not just for error conditions. This flexibility, as we shall see soon, leads to simpler and more efficient programs.

### 3.4.1 REASONS TO PROVIDE EXCEPTIONS

The main reasons, as specified by Goodenough [9], to provide exception handling facilities in a given language are

(1) To permit dealing with an operation's impending failure ie. range failures and domain failures.

(2) To permit an invoker to monitor an operation.

To illustrate what we mean by monitoring an operation, consider a recursive subprogram for searching for an item through a data structure, say, a binary tree. Each time an item is found, an exception is raised identifying the item. The invoker of this subprogram can decide whether to get the next item or not. If so, he resumes the subprogram execution otherwise it is terminated. This can be particularly economical if the subprogram's state is preserved by the handler. Thus intermediate results can be made available without unwinding the recursion. This example reinforces the point made earlier in this section that exceptions are not connected with error conditions only.

### 3.4.2 REQUIREMENTS TO IMPLEMENT EXCEPTION MECHANISM

In order to provide such an exception mechanism in Ada, the language should be able to

(1) associate handlers with invocation of operators.

(2) support default exception handling.

(3) realize various control flows during the execution of exceptions.

In the following sections, we will illustrate the desirability of these requirements through examples. Interested readers can refer to Goodenough [9] for implementation aspects. We will also cover the exception handling as it exists in current Ada [19] and try to identify its limitation and suggest some improvements.

3.4.2.1 ASSOCIATION OF HANDLERS WITH OPERATORS

A handler is associated with the point of activation of an operation, which can be either a subprogram or system defined operator, i.e., an arithemetic, boolean or relational operator. This implies that different handlers can be associated with a given operation at different points of activation. A few problems that can arise in raising an exception are

(1) Associating a handler with a wrong activation point.

(2) Associating a handler with a wrong exception. This is possible when an operation is allowed to raise more than one exception.

(3) Forgetting that an operation can raise an exception thus not providing a handler.

These problem can be tackled by letting the compiler detect these.

3.4.2.2 DEFAULT EXCEPTION HANDLERS

It is not quite often that the invoker of an operation wants to associate a different handler with each activation point.

Thus it is convenient to be able to specify a default handler, which will be executed unless specifically overridden. The exceptions which have default handlers are called default exceptions. The following are the desirable requirements in associating an default handler with a exception

(a) Default exceptions should be declared to be so. Thus the invoker of the operator can decide if he/she wants to override it.

(b) It should be possible to specify default handlers for programmer defined operations.

(c) Programmer defined and system defined default handlers should be treated uniformly.

(d) It should be possible to invoke default handlers explicitly as well as implicitly. This will provide a flexible control, both bottom-up and top-down, in exception handling.

3.4.2.3 VARIOUS CONTROL FLOWS

Different types of control flows are required to deal with different exception raising situations. We can visualize the following three possibilities of control transfer after the execution of an handler

(a) Terminate the execution of the exception raising operation and transfer control to the invoker. This type of situation can occur, for example, in overflow exception while performing some arithmetic operation.

Ada provides this.

(b) Resumption of the operation after the handler has completed its execution. This kind of situation can arise, for example, in reading a bad tape. If the read data has some parity error then it is not uncommon to reactivate the read command from within the handler.

(c) Leaving the decision of termination or resumption of an operation to the discretion of the handler. In this case, before terminating the operation some clean-up actions such as closing all files have to be performed. For a justification of this type of situation, consider a situation where the tape reading operation is to be terminated after few trials because of parity errors.

### 3.4.3 EXCEPTION HANDLING MECHANISM IN ADA

The author feels that the exception handling mechanism in Ada is a very simplified version of the above requirements and can be used only for a subset of all possible situations for which exceptions are desired. In brief it fails to satisfy the following requirements

(1) It does not provide any compile time checking to determine if an exception has been provided with the corresponding handler.

(2) It has no provisions to define default exception for the user defined operations. One can argue that for a exception raised by a subprogram we can define a

default handler in the body of the subprogram. The problem with this is that the default handler can not be overridden by the invoker. This defeats the basic purpose of providing user defined default exception handlers. This treatment of programmer defined default handlers is different from the language defined default handlers such as CONSTRAINT-ERROR, NUMERIC-ERROR etc. which can be overridden.

(3) It provides only one type of control flow, that is termination of exception raising operation. It is, in other words, a "controlled goto" passing no information (aside from the validity of the exception) to the handler from the statement that raises the exception and allowing no return to that statement. Ichbiah [14] provides a convincing argument by giving the example of reading a tape. In case of read errors, the tape is read again up to 10 times before giving up.

```
for I in 1..10 loop
   begin
     READ-TAPE(BLOCK);
     exit;
   exception
     when TAPE-ERROR =>
   if I = 10 then
     raise TAPE-FAULT;
   else
```

```
        BACK-SPACE;
    endif;
    end;
```

this seems to have eliminated the problem. The above approach may not be very convenient if the tape reading operation is to be provided at a number of places in a program because the whole structure has to be repeated. One can argue in favor of providing the tape reading operation as a subprogram with a corresponding handler associated to its body to eliminate the problem. But then it is unreasonable to provide a subroutine for every operation that can raise an exception since exception conditions occur but rarely. Such a approach will affect the run-time efficiency adversely.

### 3.4.4 SUGGESTIONS TO IMPROVE THE EXCEPTION MECHANISM IN ADA

In this section an attempt will be made to improve the exception handling mechanism in Ada in the light of the above three requirements. In accordance with the objective of this thesis, we will achieve this without introducing much complexity in the language design and with little changes in its syntax.

#### 3.4.4.1 COMPILE TIME CHECKING

The exception handling syntax in Ada avoids the first two problems mentioned in section [3.4.2.1] by associating a named handler with the program unit in which exception

raising operation is invoked. According to the author the third problem can be solved by requiring the explicit declaration of all exceptions an operation can raise and their static association with corresponding handlers. For example we can have a subprogram declaration of the following form in Ada

```
procedure EXAMPLE (formal params) [EXCEP-A: exception type]
is
--exception type refers to three different control flows
-- discussed earlier.
-- EXCEP-A refers to the exception raised by the
-- execution of this procedure.
-- local declarations
begin
  -- subprogram body
end;
```

The compiler is now in a position to check if the handler is provided in the body of invoker and warn the programmer otherwise.

### 3.4.4.2 IMPLEMENTATION OF VARIOUS CONTROL FLOWS

One of the attractive extensions of Ada can be the introduction of the subprogram type. The subprogram type, as we shall see later, enables us to achieve the desired control flow in exception handling besides providing ease of programming. One may be led to believe that this will introduce considerable complexity in the language design but

this is not so, Ada, in its present form, has a good deal of complexity already built into its generic mechanism to provide a static version of subprogram values. Thus it stands to reason that such a facility can be introduced without making the design of the language any more complex. To attack the problem meaningfully, we will be combining present exception handling mechanism with the new technique to provide all three types of control flows in exception handling. The changes to be made in the language to introduce subprogram types are given in Appendix A.

Returning to control flow issue in exception handling, consider the handling of an exception in the following example

```
procedure MAIN is
ERROR: exception;
procedure EXCEPTION-RAISER is
begin
    Statement-1;
    raise ERROR;
    Statement-2;
end;
begin
    Statement-3;
    EXCEPTION-RAISER; -- procedure call
    Statement-4;
exception
    when ERROR => Statement-5;
```

```
end;
```

According to the current Ada, statements will be executed in the following sequence if the exception condition ERROR is raised.

Statement-3, Statement-1, Statement-5, Statement-4

Thus execution of statement-2 has been skipped, whereas we would like the control to return to procedure EXCEPTION-RAISER from the handler. In this case the execution sequence will be

Statement-3, Statement-1, Statement-5, Statement-2 and Statement-4

To achieve this effect Hilfinger suggested [13] defining a package CONDITION as follows

```
generic package CONDITION is
  subtype HANDLER-TYPE is procedure; -- procedure type
  subtype BODY-TYPE is procedure; -- procedure type
  procedure SIGNAL;
  -- parameterless procedure. Invokes dynamically
  -- innermost handler established by procedure ENABLE
  -- Has no effect if there is no handler
  procedure ENABLE(BODYPART: BODY-TYPE;HANDLER:
HANDLER-TYPE);
  -- It establishes the handler with the body
  -- and then executes the body
end CONDITION;
```

Note that an instantiation of CONDITION may not be shared among multiple tasks because of the global variable in the package body. The body of package CONDITION can be defined as follows

```
package body CONDITION is
subtype FRAME-TYPE is procedure;
CURRENTFRAME: FRAME-TYPE := begin null; end;
-- this defines the defualt action for CURRENTFRAME
procedure  ENABLE(BODYPART:BODY-TYPE;  HANDLER:HANDLER-TYPE)
is
  LASTFRAME: constant FRAME-TYPE :=CURRENTFRAME;
  procedure NEWFRAME is
    begin
      CURRENTFRAME := LASTFRAME; -- assignment of subpr.
                                 --variables
      HANDLER; -- execution of procedure HANDLER
      CURRENTFRAME := NEWFRAME;
      exception -- note exception OTHERS will be raised
                --when some exception is raised within
                -- handler and we want to terminate
      others => CURRENTFRAME := NEWFRAME;
  end NEWFRAME;
begin
    CURRENTFRAME := NEWFRAME; -- establish handler
    BODYPART; -- execution of the body
```

```
    CURRENTFRAME := LASTFRAME;
    exception -- make sure frame pointer is restored
others ⇒ CURRENTFRAME := LASTFRAME;
end ENABLE;

procedure SIGNAL is
  begin
    CURRENTFRAME; -- execution of current handler
  end;
end CONDITION;
```

Once these packages are defined, associating an exception with a handler is very straight forward process. Exception ERROR will be defined by instantiating the package CONDITION. Let us associate this exception with the program MAIN discussed earlier.

```
package ERROR is new CONDITION;
procedure EXCEPTION-RAISER is
  begin
    Statement-1;
    ERROR.SIGNAL; -- activating SIGNAL procedure in ERROR
    Statement-2;
  end;
procedure MAIN is
  begin
    ERROR.ENABLE( begin   -- calling procedure
```

```
        Statement-3; -- ENABLE in package ERROR
        EXCEPTION-RAISER;
        Statement-4;
        end;
        HANDLER => begin Statement-5;
                   end;);
  end;
```

Note that the original main program body and the associated handler are passed to the procedure ENABLE as actual parameters. Let us trace how the exception ERROR will be handled using the above approach.

The execution starts with a call to procedure ENABLE. Declaration LASTFRAME is elaborated and it is initialized to procedure value

```
begin null; end;
```

After the elaboration of declarations of procedure ENABLE, the execution of first statement assigns a value

```
begin
  CURRENTFRAME := LASTFRAME;
  Statement-5;
  CURRENTFRAME := NEWFRAME;
  exception
    others => CURRENTFRAME := NEWFRAME;
end;
```

to variable CURRENTFRAME.

Execution of BODYPART procedure within the body of procedure

ENABLE implies the execution of Statement-3, then a call to procedure EXCEPTION-RAISER, and finally Statement-4. Within the procedure EXCEPTION-RAISER Statement-1 is executed and exception ERROR is raised by activating the procedure SIGNAL, which executes the procedure CURRENTFRAME. Thus Statement-5 is executed and control is returned back to procedure EXCEPTION-RAISER. Note this method is recursive so the nesting can be as deep as permitted by the stack size constraint.

To resolve the third and the final control flow issue, that is, making the resumption of the exception raising operation conditional, the author suggests the following
Suppose in the procedure EXCEPTION-RAISER we wish to terminate its execution if a certain condition is met in the handler. To achieve this the variable HANDLER can be modified as

```
declare
  EXIT-EXCEP: exception;
  begin
    Statement-5;
    if (condition) then raise EXIT-EXCEP;
  end;
```

and the procedure NEWFRAME will be modified as

```
procedure NEWFRAME is
  begin
    CURRENTFRAME := LASTFRAME;
```

```
    HANDLER;
    CURRENTFRAME := NEWFRAME;
  exception
    others = CURRENTFRAME := NEWFRAME;
        raise; -- raises the exception again
  end;
```

So the exception EXIT-EXCEP will be raised again in procedure EXEPTION-RAISER. Now some cleanup operation, if desired, can be provided within the procedure EXCEPTION-RAISER and then the control will be transfered to the main program.

In summary, the author would like to emphasize again that the Ada exception mechanism can be improved, with very little changes in the language design, to provide

(1) compile time checking of association of exceptions with corresponding handlers.

(2) flexible control flow.

However, default exception handling still remains an issue to be resolved.

# CHAPTER FOUR

## CONCLUSION

In this thesis, the following three questions were raised

(1) Why is a language as complex as Ada desirable for programming?

(2) How can the design of Ada be simplified without compromising its power?

(3) How can the capabilities and expressive power of Ada be improved without affecting the complexity of its design significantly.

We answered these questions in sequence by

(1) showing the superiority of the abstract mechanisms in Ada over PASCAL.

(2) suggesting some simplifications in Ada. In particular, we emphasized on

(a) removal of derived types since their functionality can be achieved using existing features in the language thus simplifying the implementation of the language.

(b) restricting the shared variables and tasks sharing them to reside on the same computer oin a

multicomputer architecture. This simplified the design of the compiler by eliminating the generation of communication protocols and thereby improving the run-time efficiency.

(3) introducing new abstractions in the language. In particular, we saw that

(a) the unidirectional message communication between tasks can be implemented more efficiently by providing a implicit buffer with each entry point.

(b) redefinition of some primitive operators such as the assignment operator can make the language more expressive and functional.

(c) implementation of an abstract "deschedule" would facilitate the design of time-sharing environments.

(d) subprogram types enabled us to implement the exception handling in a very flexible way.

The author would be wrong to claim that these are the only problems with Ada and these are the only solutions. The implication of the study is that there is lot of room for improvements in Ada and it can be achieved without changing the present Ada's design significantly.

# REFERENCES

[1] Barnes, JGP, Programming in Ada, International Computer Science Series, 1982

[2] Brinch-Hansen, P., The Architecture Of Concurrent Programs, Prentice-Hall Inc., Englewood Cliffs, New Jersey 07632, 1973

[3] Brinch-Hansen, P., Operating Systems Principles, Prentice-Hall Inc., Englewood Cliffs, New Jersey 07632, 1973

[4] Brinch-Hansen, P., 'Distributed Processes: A Concurrent Programming Concept', Communications of the ACM, Vol. 21, Number 11, pp 934-941, Nov 1978

[5] Dijkstra, EW, 'The Humble Programmer', Turing Award Lecture, Communications of the ACM, Vol. 15, Number 10, Oct 1972

[6] Downes, V. A., Goldsack S. J., Programming Embedded Systems With Ada, PrenticHall International, Inc., 1982

[7] Gehani, N., Ada An Advanced Introduction, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1982

[8] Gehani, N., 'Concurrency in Ada and Multicomputers', Computer Languages, V7, No.1, 1982

[9] Goodenough, John B., 'Exception Handling: Issues and a proposed notation', Communications of the ACM, Vol.18, Number 12, pp 683-696, December 1975

[10] Goodenough, John B., 'The Ada Compiler Validation Capability', IEEE Computer, pp 57-64, June 1981

[11] Hoare, CAR, 'Communicating Sequential Processes', Communications of the ACM, Vol. 21, Number 8, pp 666-676, August 1978

[12] Hilfinger, Paul N., 'Simulation of Procedure Variables Using Ada Tasks', IEEE Transactions on Software Engineering, Vol. SE-9, No. 1, January 1983, pp 13-15

[13] Hilfinger, Paul N., 'Abstract Mechanisms and Language Design', Ph.D, Thesis, Department of Computer Science, Carnegie-Mellon University, June 1981

[14] Ichbiah, J., View-Graphs for Jeans Ichbiah's Presentation in 'Proceedings of the Ada Debut', Defense Advanced Research Project Agency, Arlington, VA 22209, Sept. 1980

[15] Leblanc, J. Richard, Goda J. John, 'Ada and Software Development Support: A New Concept in Language Design', IEEE Computer, May 1982, pp 75-83

[16] Mben Ari., 'Principles of Concurrent Programming', Prentice-Hall International, Inc., 1982

[17] Nassi, R. Isaac, 'What is Ada', IEEE Computer, pp 17-24, June 1981

[18] Pyle, I. C., The Ada Programming Language, Prentice-Hall International, Inc., 1982

[19] Reference Manual for the Ada Programming Language, Honeywell, Systems & Research Center, January 1983

[20] Shaw, Mary, Studies In Ada Style, Springer-Verlag,

1981

[21] Shaw, Mary, 'The Impact of Abstraction Concerns On Modern Programming Language', Proceedings of the IEEE, Vol.68 Number 9, pp 1119-1130, September 1980

[22] Singer, Andrew, Ledgard F. H, 'Scaling Down Ada (or towards standard Ada)', Vol. 25, Number 2, pp 121-124, February 1982

[23] Stenning, Vic 'The Ada Environment: A Perspective' IEEE Computer, pp 26-36, June 1981

[24] Wegner, Peter, 'Self Assessment Procedure VIII', Communications of the ACM, Vol 24, Number 10, pp 647-676, October 1981

[25] Wegner, Peter, Programming With Ada: An Introduction By Means Of Graduate Examples, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1980

[26] Welsh, Jim, and Lister, Andrew, 'A Comparative Study Of Task Communications in Ada', Software-Practice and Experience, Vol. 11, pp 257-290, 1981

1. The syntax for type_mark is changed to

```
type_mark ::= type_name | subtype_name | subprogram_type
```

2. A subprogram_type is essentially a subprogram specification, slightly modified:

```
subprogram_type ::=
    procedure [ formal_part ]
  | function [ formal_part ] return subtype_indication
```

3. The operation ":=" is defined implicitly on each functional type, but "=" is not. The effect of the assignment is to make the assigned-to variable refer to the subprogram value of the right operand, but with parameter names and default values as given for the declaration of the left operand. This same replacement of parameter names and default values occurs to a subprogram value returned as the result of a function call.

4. A named constant of a subprogram type may be deferred, just as can a constant of a private type[38]. A complete definition must appear later in the same declarative part (where, for this purpose, the body and private part of a package are considered to be in the same declarative part as the visible part).

5. The literals of a subprogram type are unlabeled blocks:

```
subprogram_constant ::=
            [declare
              declarative_part]
            begin
              sequence_of_statements
            [exception
              {exception_handler} ]
            end
```

The same block may designate subprograms of several different subprogram types. The type must be determined from surrounding context, without reference to the text of the literal. Any free variables in the subprogram constant are bound in the context in which the constant is elaborated (i.e., not at the point the subprogram denoted is eventually called). Note that the rules stated so far combine with existing parts of the language to provide fully qualified subprogram constants (what a Lisp programmer would call lambda expressions), as in

```
function (x: INTEGER) return INTEGER'(begin return x+1; end)
```

which denotes the successor function on the type INTEGER.

6. The declaration

```
procedure identifier [ formal_part ] is
      declarative_part
begin
   . . .
end [ designator ];
```

is an alternate form of

```
identifier: constant procedure [ formal_part ] :=
                  declare
                     declarative_part
                  begin
                     . . .
                  end [ designator ];
```

and likewise for functions.

7. Two instances of subprogram_type denote the same type if the sequence of subtypes of the formal parts (in order) are pairwise identical, and the return types, if any, denote the same subtype.

8. Calling a subprogram object that has an assigned value proceeds as does a call in current Ada. Calling a subprogram object without a defined value raises an exception (UNDEFINED_SUBPROGRAM). As in current Ada, names of subprogram objects may be overloaded and resolved by context.

9. It is illegal to call a subprogram value when an object referenced in that subprogram value has been de-allocated.

10. Subprogram designators (such as "+") are legal as identifiers in object declarations (allowing, e.g.,

```
":=" : procedure (x: out T; y: T);)
```